Predicting Download Directories for Web Resources

George Valkanas
Dept. of Informatics and Telecommunications
University of Athens
Athens, Greece
gvalk@di.uoa.gr

Dimitrios Gunopulos
Dept. of Informatics and Telecommunications
University of Athens
Athens, Greece
dg@di.uoa.gr

ABSTRACT

Browsing the web is one of the most common activities that users engage in nowadays, and downloading web resources of interest, such as images, documents, music, etc., is part of this process. However, users would rather temporarily save that resource to a default path that they have easy access to (e.g. their "Desktop") than select the actual directory where they would eventually place it. This clearly implies that existing user interfaces are not as effective for this particular task as the users would like them to be. Instead of proposing a different User Interface, in this paper, we try to address the problem at its core, and propose a methodology to suggest the most likely directory where the file would (eventually) be saved by the user. By doing so, future interfaces can also benefit from our technique. We provide a formal definition of the problem and propose a classification framework to tackle it. We present our overall solution to this problem, namely *Di*rectory *Do*wnload Predi*Ctor*, or *DiDoCtor* for short. We give experimental evidence of its effectiveness, by implementing our approach as part of a widely used browser and evaluate it with real user activity. We also discuss lessons learned from this process, regarding the efficiency perspective.

Keywords

Web Browsing; Directory Prediction; UI assistance

Categories and Subject Descriptors

H.5.3. [**Information Interfaces and Presentation (e.g. HCI)**]: Group and Organization Interfaces

General Terms

Human Factors; Algorithms; Management

1. INTRODUCTION

Numerous surveys and statistic results reveal that computers are used in almost all aspects of everyday life [1, 32, 38] and that a lot of this time is spent on internet activities. Searching for information and using email still seem to be the most prominent aspects of computer usage on the web, although social media interaction is constantly taking rise. With the wealth of information that is currently available, provided under a wide range of formats and technologies, web browsers have evolved, out of a necessity to stay competitive and maintain or increase their market share. Browsers have, thus, changed from simple, passive html renderers, to elaborate software applications with advanced functionalities such as caching, private browsing, and proactive behavior, such as link prefetching [22]. It is clear that browsers are one of the most daily used programs in a computer and are basically a primary component in interacting with web related content.

Similar surveys [1, 2, 25] have also shown that a usual activity of internet users is to download content from the web. This includes downloading software, images, documents, music from online stores, etc. Additionally, email still remains the most popular activity, and email-related surveys [3] show that a high percentage of emails contain attachments. Users save these resources to directories on their computers, in a way that keeps them contextually organized. Although this is a casual web browsing activity, it has been mostly overlooked and few steps have been taken so far to facilitate it. The standard practice that most, if not all, browsers follow when a user downloads a file is to request from the user to specify the directory[1] where it should be placed, displaying the well-known *File Chooser* user interface (UI). In the best case scenario, the *File Chooser* UI will point to the path that was last used for saving another resource, whereas another usual approach is to select one of a predefined set of directories, based on the type of the downloaded file (e.g., image, document, music, video, etc.). Alternative UI-based approaches have also been implemented, all as browser extensions [4, 5, 6], which use cascading menus, as a replacement to the *File Chooser*. The thousands of downloads of these plugins vividly demonstrate the need for better ways to perform the process of saving resources.

However, all of these approaches are rather impractical. More specifically, the last used location may be completely unrelated to the current file, because e.g., there is no temporal or contextual coherency between them. Similarly, using a fixed set of predefined directories, means that we disregard entirely the characteristics of the file that is being downloaded. What is more, users typically construct several folders, many of which are outside the predefined ones, essentially negating any advantage that this approach may have had. Finally, the latter case only offers an alternative UI, but does not tackle the problem of locating a good candidate directory at its core. Therefore, it suffers from the same inefficiencies as the

[1]A directory is the naming equivalent of a folder for *-nix systems. For our purposes, there are no differences between the two and we use the terms interchangeably.

WIMS'14, June 2-4, 2014 Thessaloniki, Greece.

previous two. Moreover, it results in UI clutter, due to several cascading menus appearing all at the same time, which also obstruct a large portion of the user screen.

Whatever approach is being used, including that of cascading menus, the user is required to navigate to the right directory every time, where they wish to save the file. Despite taking only a couple of seconds, the selection of the proper download directory may easily throw the users out of context of their current web task. To avoid this, users tend to download their files to an easily accessible directory, e.g., "Desktop" or "Downloads", and move it to the desired one afterwards. Evidently, this is a tedious task for the user, one they would like to avoid overall. We can thus safely argue that response and interaction times are crucial factors for user satisfaction, much like other UI-related processes [14, 16, 36]. Automatically identifying the right directory to download a file would then greatly improve the user experience.

In this paper, we tackle the problem mentioned above, namely how to efficiently and effectively identify the directory where a user would save a file they wish to download. By *file* we mean any type of resource which can be locally saved on the user's machine. In particular, in this work, we make the following contributions:

- We provide a concrete problem formulation, viewing it as an optimization one.
- We cast the original problem to a classification framework, to solve it efficiently, resulting in our *DiDoCtor* approach. We introduce the appropriate concept mappings and identify applicable techniques, given the particularities of the setting (time constraints and dynamic setting).
- We present an extensive evaluation on the efficiency and effectiveness of *DiDoCtor*, using widely acknowledged measures. Our experiments are based on real-world deployments of our implementation, with actual user involvement.

Where appropriate, we also briefly discuss our experiences and lessons learned from implementing our approach as a plug-in for a well-known, extensible browser.

The rest of the paper is organized as follows. Section 2 discusses related work on the subject and similar topics. Section 3 introduces our problem formulation, followed by Section 4 which provides both the general approach and the techniques in particular that we implemented. Section 5 presents our detailed experimental evaluation, followed by Section 6 which concludes the paper.

2. RELATED WORK

Automatic categorization of documents has always been a topic of active research, in various settings and domains. Approaches related with hierarchical structures also exist [15, 20], however, they do not pose the same constraints that we do, i.e., real-time response, dynamic environment that requires constant updating of classes and classifiers, and minimal resource consumption, to name a few.

Applications on email organization such as [19, 34] are the most relevant. The work in [19] classifies emails into activities, using conversational and reply-back features, but does not consider hierarchies. MailCat [34] aids its users to organize their email in archives based on their textual content. Though the ultimate goals, i.e., document organization, largely overlap, the domains of application make the actual problem different. For instance, emails are text-based making classic TF-IDF weighting schemes applicable, and involved people (e.g., sender and recipients) are high-quality features. On the contrary, we operate in a heterogeneous, web-based environment. Resources such as images and videos are by definition non text-based, which makes text-driven approaches unsuitable. Moreover, classifying emails can be – and are – performed in the background because the user is unaware of when an email is received. However, suggesting a download directory is triggered by actions performed by the user, therefore the classification process really needs to run in real-time.

Suggesting folders to locate files has also been the purpose of the *FolderPredictor* system [9]. The target goal is to minimize the number of clicks to the directory that the user will eventually select, where the file of interest is located. The advantage of the *FolderPredictor* approach is that it is not restricted to a particular application. However, in doing so, the system can not use high quality task-specific features, as we do. For the same reason, *FolderPredictor* operates under the fundamental assumption that the users will break down their actions into discrete *tasks*, so that the system can learn the properties related with this task. Therefore, the first time a directory is used, the user must associate it with a task, to train the system. Such a process is known to be impractical [35], and no user would be willing to undergo such a process, when in fact they try to avoid a procedure (selecting a directory) that requires even less effort. As this requires additional involvement from the user's part, they would simply revert to the downloading files to the default location. Moreover, today's web browsers promote multitasking, by allowing multiple tabs to be open with content from various sources, which contradicts the *single-task* assumption of *FolderPredictor*.

Put in a broader context, our goal is to organize web resources in hierarchical structures in a principled manner. Therefore, the work presented in [37] can be seen as relevant, where entire websites are organized in large online directories, like the Open Directory Project[2]. Such services organize web pages by leveraging Subject Hierarchies, usually a taxonomy or an ontology. These approaches are not limited by real-time constraints, whereas our technique is part of a User Interface, triggered by user actions, and response time is very crucial. Should this process take long enough, the user will resort to the predefined folder solution, negating any advantage our method may have had. Secondly, we do not rely on external information, i.e., Subject Hierarchies, whatsoever. As demonstrated by our evaluation, users organize their files differently, therefore a single global hierarchy would be ineffective. Last but not least, such approaches assume a static set of directories, i.e. all paths in the hierarchy are known in advance and do not change. On the contrary, we are dealing with a dynamic environment, where directories are added and deleted at any time. This trait adds a temporal dimension, which is not present in web directory orchestration.

Finally, several technical solutions exist [5, 7, 6] with the same goal in mind. Such approaches rely on user-defined filters, typically using the file type and domain name of the resource. The file is then saved to the first directory for which the criteria hold. If none of the filters apply, a default directory or the last download directory is used. However, these approaches are restrictive, much like the ones undertaken by modern browsers: they only faciliate folders for which a filter has been created. Since users download files to several directories, the filter construction process easily becomes tedious, and the default location becomes the norm. Additionally, web portals or sites with a broad topical spectrum (e.g., social networks, web-based email, etc.) are a bad fit for these alternatives: their content largely varies but it will always be saved to the same directory. On the contrary, no user setup is required by our approach, meaning that it can be integrated into any browser. The only user input we expect is the selection of the final down-

[2] `http://www.dmoz.org`

load directory. We also consider a much richer feature space, to differentiate between items from sites with varied content.

3. PROBLEM FORMULATION

In this section, we present a formal definition of our problem, but first we introduce some necessary notation. We assume a hierarchical structure $\mathcal{H}$, e.g. a computer's file system, like the one shown in Fig. 1 [3]. A user navigates through $\mathcal{H}$ by moving *up* or *down* one directory at a time.

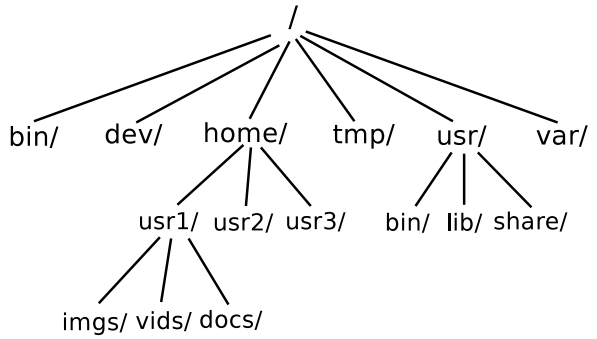


Figure 1: An example of a file system's hierarchical structure

A path $\mathcal{P} \in \mathcal{H}$ is a sequence of directories to which the user may navigate [4]. The length of $\mathcal{P}$, $len(\mathcal{P})$, is the distance of the furthest directory of $\mathcal{P}$ from the root node. For instance,

$$len(\text{“}/usr/bin/\text{”}) = 2$$

$$len(\text{“}/home/user1/imgs/Vacation/\text{”}) = 4$$

The *Lowest Common Ancestor*, LCA, of $\mathcal{P}1$ and $\mathcal{P}2$ is a path $\mathcal{P}3$ of maximal length, from which both $\mathcal{P}1$ and $\mathcal{P}2$ can be derived. In other words, the LCA is the longest common prefix of the two paths in the hierarchy. For example,

$$LCA(\text{“}/usr/bin/\text{”}, \text{“}/usr/lib/\text{”}) = \text{“}/usr\text{”}$$

$$LCA(\text{“}/usr/bin/\text{”}, \text{“}/home/user1/imgs/\text{”}) = \text{“/”}(root)$$

Using the *Lowest Common Ancestor* we can define a *descendant* relation between two paths.

DEFINITION 1 (DESCENDANT PATH). *A path $\mathcal{Q}$ is a descendant of path $\mathcal{P}$ iff LCA($\mathcal{Q}$, $\mathcal{P}$) = $\mathcal{P}$.*

Basically, the definition states that a path $\mathcal{Q}$ is a *descendant* of path $\mathcal{P}$ if and only if $\mathcal{Q}$ is derived from $\mathcal{P}$, by appending any number of directories to it (to $\mathcal{P}$). Since $\mathcal{P}$ is a common prefix of the two paths, it holds that LCA($\mathcal{Q}$, $\mathcal{P}$) = $\mathcal{P}$.

We can then define the cost of navigating from path $\mathcal{P}1$ to path $\mathcal{P}2$ as follows:

DEFINITION 2 (HIERARCHICAL NAVIGATION COST). *Navigating from path $\mathcal{P}1$ to $\mathcal{P}2$ in the hierarchy $\mathcal{H}$ incurs the cost of moving from $\mathcal{P}1$ to LCA($\mathcal{P}1$, $\mathcal{P}2$) and then to $\mathcal{P}2$, i.e.*
HNC($\mathcal{P}1$, $\mathcal{P}2$) = cost($\mathcal{P}1$, LCA($\mathcal{P}1$, $\mathcal{P}2$)) + cost(LCA($\mathcal{P}1$, $\mathcal{P}2$), $\mathcal{P}2$)

[3]The figure is based on *nix file systems, but the formalism is independent of a file system

[4]Until now, we have silently used “directory” as a shortform of the full associated path.

Several costs could be employed for navigating between paths in the hierarchy $\mathcal{H}$. For our setting, we assume that moving up or down a directory both have the same cost c. This assumption is driven by modeling simplicity and the fact that actual navigation is commonly performed in the same way (a “double click”), regardless of directionality (epsecially through a *File Chooser* UI). Therefore, the total cost of navigating from a path $\mathcal{P}$ to a descendant path $\mathcal{Q} \in \mathcal{H}$ is simply the length of their non-common part, or simply put

$$cost(\mathcal{P}, \mathcal{Q}) = len(\mathcal{Q}) - len(\mathcal{P})$$

LEMMA 1. *For any two paths $\mathcal{P}$, $\mathcal{Q}$, where $\mathcal{Q}$ is a descendant of $\mathcal{P}$, it holds that cost($\mathcal{P}$, $\mathcal{Q}$) = len($\mathcal{Q}$) - len($\mathcal{P}$) $\geq$ 0.*

Since $\mathcal{Q}$ is a descendant path of $\mathcal{P}$, by definition $\mathcal{Q}$ has been derived by appending any number of directories to $\mathcal{P}$. If $\mathcal{Q}$ = $\mathcal{P}$, i.e. no directories have been appended, then cost($\mathcal{P}$, $\mathcal{Q}$) = 0. Otherwise, at least 1 directory has been appended to $\mathcal{P}$, therefore len($\mathcal{Q}$) > len($\mathcal{P}$) and the inequality surely holds. For the rest of the discussion we assume this cost function.

We can now define the download directory suggestion problem as an optimization one, whereby we try to minimize the hierarchical navigation cost from the directory we suggest to the directory that the user will eventually select.

PROBLEM 1 (DOWNLOAD DIRECTORY SUGGESTION). *Let $\mathcal{H}$ be a hierarchical structure, i.e., a computer's file system, and let $\mathcal{T} \in \mathcal{H}$ be the target path where the user will eventually download the web resource. We want to suggest a directory (i.e., full path) $\mathcal{S} \in \mathcal{H}$, such that $\mathcal{S}$ minimizes the hierarchical navigation cost from $\mathcal{S}$ to $\mathcal{T}$, i.e.,*

$$\arg\min_{\mathcal{S} \in \mathcal{H}} HNC(\mathcal{S}, \mathcal{T})$$

Being a UI related problem, where all actions are triggered by the user (unlike the email scenario), there is a hard constraint on real time execution of the suggestion process, so that the user receives immediate feedback.

Evidently, if we could suggest $\mathcal{T}$, i.e., $\mathcal{S}$ = $\mathcal{T}$, we obtain an optimal solution to Problem 1, because then cost($\mathcal{T}$, $\mathcal{T}$) = 0. However, the problem is that we do **not** know path $\mathcal{T}$, until *after* the user has actually selected it.

To confront this shortcoming, we rely on properties, mainly of the web resource, in order to suggest an appropriate path $\mathcal{S}$. To that end, we can cast the *Download Directory Suggestion* problem to a classification framework, without changing our optimization goal, in the following way:

- Paths correspond to class values.
- Path $\mathcal{T} \in \mathcal{H}$ is the true target class.
- Path $\mathcal{S} \in \mathcal{H}$ is the outcome of the classification process.
- Properties of the web resource are features that we utilize for the classification process.

Under this framework, we want to find and suggest the class $\mathcal{S}$ that best ressembles $\mathcal{T}$, where the web resource will finally be saved. We can now rely on prior knowledge and properties of web resources that the user has already downloaded. Note that the click minimization goal is a latent one here: all the more similar $\mathcal{T}$ is with $\mathcal{S}$, all the fewer clicks will be required to navigate from the one path to the other.

3.1 Classification Framework Issues

Although casting the optimization problem to a classification one helps addressing it, it also raises some issues we need to tackle. First of all, it is clearly inefficient to consider every path in $\mathcal{H}$ as a possible class. A vanilla Operating System (OS) installation contains several hundreds, maybe thousands of directories and considering all of them as classes is a waste of resources: most are related to the OS itself and default users do not have permission to save files there. Moreover, as our experimental evidence show, illustrated in Figure 2, even highly selective users tend to use a rather small set of directories to save their work compared to $\mathcal{H}$.

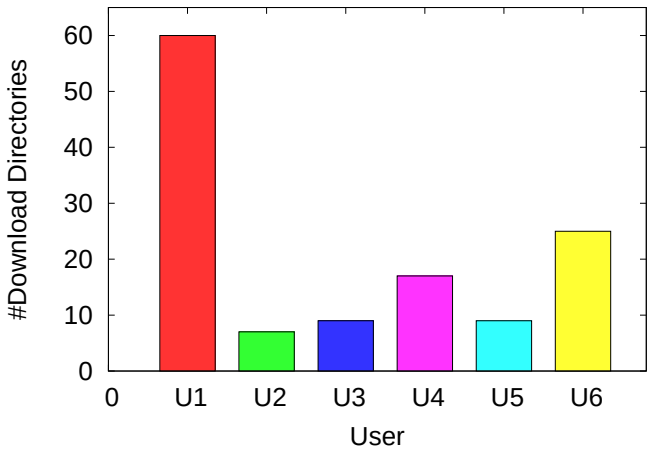


Figure 2: Number of directories used to download resources by each user in our experimental dataset

On another note, we feel that scanning the file system does not only take a considerable amount of time – even if that occurs once at browser startup –, but it is also quite intrusive. Some browsers allow for "*Private Browsing*", meaning that no information is collected during that period. Private browsing promotes additional safety, especially when a user deals with sensitive data (e.g. tax agencies) and obtaining such directories by scanning the file system negates the entire idea.

Thirdly, users create and delete directories ad-hoc, to contextually organize their files. Therefore, the classification process should be able to handle a dynamic environment, with impromptu class additions / deletions. Basically, this characteristic implies that training the classifier i) should *not* be time consuming, so that any class changes are promptly taken into account and ii) it should go unnoticed by the user, without any UI freezes or performance slow downs in general.

Finally, each user organizes their files a lot differently than others. Therefore, we can not make any assumptions regarding the structure of the hierarchy $\mathcal{H}$, other than its tree form.

4. THE DIDOCTOR APPROACH

Given the general classification framework that we described in the previous section, we now discuss how these ideas are materialized in practice. In particular, we have integrated the discussed concepts in our *Di*rectory *Do*wnload Predi*Ctor* – *DiDoCtor* for short – approach.

4.1 Feature Selection

We start by briefly introducing the features that we have considered in our implementation of the classification framework, and proceed to justify them afterwards. Selecting a good set of features is important because it directly affects the classifier's accuracy. We also need to bear in mind that the features must be easy to extract, both in terms of time- and memory-related resources, so that we do not degrade the user experience.

- **Timestamp** when the download has been requested.
- **Domain name**, **path** and **filename** (as distinct features) of the page containing the web resource to download.
- **Domain name**, **path** and **filename** (as distinct features) of the *referrer* page. The referrer page is the one from which the user landed on the current page.
- **Title** of the page containing the web resource to download, extracted from the HTML <*title*> tag.
- **Filename** of the web resource the user is downloading.
- **Extension** of the web resource the user is downloading, e.g., pdf, jpeg, zip, etc.
- **Keywords** extracted from near the link of the web resource we are downloading.

The **timestamp** is used to capture temporal dynamics of a user's downloads. For instance, some downloads might exhibit periodicity. More importantly, temporal information may identify *sessions* of user activities. Borrowing from web server terminology, a *session* is a period of time (typically 20 minutes) during which a user interacts with a web server under the same context. Therefore, it makes sense to consider similar periods during which resource downloading is highly contextualized. From the timestamp, we can also extract temporal information such as day of the week or month.

The **domain name**, **path** and **filename** of the web page where the resource can be found may be good descriptors of what the resource is about. Note that the **domain name** alone is already used by browsers, on the basis that any item downloaded from a specific domain (e.g., `http://iuiconf.org`) will be saved to the same directory where other resources were saved from that domain.

The idea behind using information from the *referrer* page is twofold: i) it is known that the web exhibits topical locality [17], meaning that web pages that are hyperlinked tend to have similar content, and ii) users rely on search engines to find information on the web, in which case the query terms become part of the referrer link. Since the user has already left the referrer page, the referrer link is the most easily accessible information, because it is part of the HTML headers exchanged with the server.

Using the **title** of the web page comes from the importance of the element itself. Quoting from W3C's HTML 4 specification [8], "*every HTML document **must** have a TITLE element in the HEAD section*", and "*authors should use the TITLE element to identify the contents of a document*". It is then only natural that we use the content of this element as a feature.

The **filename**, much like the name of the current page, can provide useful insights on the item in consideration.

The **extension** of a file define its content (e.g. "jpg" for images, "avi" for video, "txt" for simple text, etc.), and contemporary operating systems rely on it even today for this very purpose. As the extension is a discriminating factor, one easily perceived by human readers, it could improve classification accuracy all the same.

Finally, we extract **keywords** related with the resource. The keyword from the anchor text of the hyperlink that points to the resource of interest. The anchor text is the text found inside and surrounding the *anchor* element (<*a*>). Its significance is known since the mid 90s [29], and has improved search queries [21], text classification [23] and scalable web document clustering [31] to

name a few. The rationale is that the anchor text is a good descriptor of the resource that the hyperlink points to, so it makes perfect sense to use it to extract such keywords.

4.2 Feature Distances

The features we use have different semantics. For example, the extension gives a general view of the content type of the resource, e.g., "video", "image", etc., whereas the title and keywords reveal some information on the content itself, e.g., "cartoon movie" or "landscape picture". Basically, this means that various distances can be employed, and some may be more appropriate than others for each feature.

For instance, Jaccard distance on web page keywords has been a proven technique for near-duplicate web page detection [12]. On the other hand, the edit-distance on the resource filename or simple equality checks on the domain name could yield better results. Finally, we could consider a covariance distance matrix for filetypes. Therefore, two documents with "doc" extension have distance 0, whereas a document with "doc" and one with "pdf" extension have an extension distance higher than 0. However, two documents, one with a "doc" extension and one with an "mp3" extension seem far less relevant than the pair ("doc", "pdf"). Such a matrix could be hardcoded or could be learned from user data.

In the implementation for our evaluation, for each of the discussed features, we used the distances shown below. Note that we are given the information of the current item, and we compare it against previously stored items, for which this information has also been extracted and stored locally on the user's system.

- **Timestamp**: We used an exponentially declining similarity, of the form $\exp^{-\lambda(t_2-t_1)}$. The difference $t_2 - t_1$ measures the time elapsed (in seconds) between the current timestamp t_2, i.e., when the resource is being stored, and the timestamp t_1 when another resource was saved. The factor λ accounts for the sessions, and is equal to $\lambda = \frac{1}{1200}$, i.e., the number of seconds in a 20 minute period (so that both are properly scaled). Recources which have been more recently saved will have a higher similarity, especially those within the 20 minute period (sessions). Finally, because this is a similarity, we use $1 - \exp^{-\lambda(t_2-t_1)}$ as the respective distance.
- **Domain name**, **path** and **filename** of both the current page and the referrer page. For the **domain name** we check whether the two domains are the same, returning 0 if they are equal, or 1 otherwise. We split the **path** and compute their Jaccard distance. Note that more complex techniques could be employed for both the domain and path [10] but we do not consider them here. Regarding the **filename**, we also tokenize it and compute the Jaccard distance between the current resource and the potential match.
- **Title**: We also use the Jaccard distance of the pages' titles, after tokenization and normalization (e.g., lowercasing).
- **Filename**: Filenames of the downloaded resources are compared with the Jaccard distance, after tokenization and normalization (e.g., lowercasing).
- **Extension**: For this feature, the distance is computed using a hardcoded covariance matrix. If the two extensions are the same, the distance is 0. If they are different, we get the general type of each extension. For instance, the type of "jpg" is "image", whereas for "html" it is "web-page". If the two extensions fall under different categories (e.g., "music" and "document"), the distance is 1. If they fall under the same category, we return 0.5. We can also have more fine-grained information at this stage. For example, extensions "doc" and "docx" should be closer than "doc" and "pdf", but both cases should return a value lower than 1.
- **Keywords**: The distance between the two sets of keywords is given by their Jaccard distance.

Finally, the total distance between the two items is given as a weighted sum of the distances that we mentioned above. We provide more details regarding the weights in a subsequent section. Feature distances were computed independently of the other features, i.e., filenames and keywords were in separate bag-of-words, over which we computed the Jaccard.

4.3 Identifying target classes

We have already discussed the impracticality of scanning the computer's file system to extract target classes. Instead we maintain a set of directory paths where the user has downloaded a web resource at least once in the past.

Moreover, since a computer's file system is inherently organized in a hierarchical structure, we could also view the problem as one of hierarchical classification [20]. This does not change neither the features we use, nor the target goal. It provides us, however, with the possiblity to consider intermediate directory paths as candidate target classes. This approach is useful when we are not entirely certain of the exact target class, yet suggesting a higher level directory could minimize the number of clicks on average.

5. EXPERIMENTAL EVALUATION

We implemented our *DiDoCtor* approach as a plugin (*add-on*), for the Mozilla Firefox browser. The browser decision was based on its open source nature, its wide adoption, and its extensibility capabilities, as exemplified by the approximate running average of 450M plugins used during April 2013 [5].

The plugin was fully implemented in JavaScript, the default language for Firefox add-ons. It came with a weighted 1-NN classifier, with custom weights, and we integrated all features presented in the previous paragraphs. We discuss the effect of weight selection as well. We requested from a group of approximately 50 people (friends, colleagues, relatives, etc.) that they download the plugin, install it and continue to use the browser as usual. After a usage period of 4 months minimum, the users were requested to anonymously submit their locally stored data through a web form.

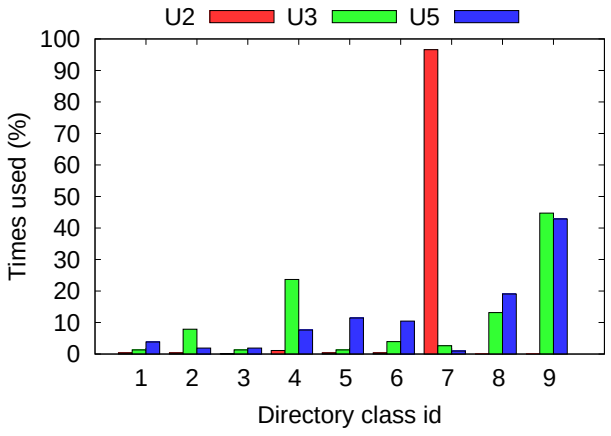


Figure 3: Directory usage histogram for 3 of our experiment's users

[5] `https://addons.mozilla.org/en-US/statistics/`

A total of 6 users voluntarily submitted their usage data, with varying access patterns: Figure 2 already demonstrated the difference in the number of directories used by each user. Figure 3 shows a path usage histogram for 3 users of our plugin. Although all of them have used at most 9 distinct paths, the usage pattern is completely different (the actual paths are also different). User U2 uses a single directory most of the time, with the rest of the directories being used only 2 or 3 times. On the other hand, users U3 and U5 exhibit a more even behavior among the directories used. Table 1 summarizes some basic statistics on the download patterns of each user, including the average path length where resources are saved, the average number of times each path is used, etc.

Table 1: Basic statistics on directory usage (per user)

User	#Paths	Path Length		Times used	
		AVG	*SDEV*	*AVG*	*SDEV*
U1	60	6.39	1.16	2.68	3.80
U2	7	4.05	0.37	37.85	89.06
U3	9	4.57	0.66	8.44	10.49
U4	17	5.63	1.58	10.23	13.19
U5	9	4.99	0.92	11.66	13.14
U6	25	4.93	1.40	4.08	7.80

As a baseline comparison, we use the current standard approach employed by most web browsers: for every domain name, we maintain the directory path that was used last to download a resource. Whenever a new download is initiated, the user is prompted to the last directory where a resource was saved from that same domain. For new domain names, a default directory is used (e.g. "Downloads"). Therefore, this approach completely neglects content, extensions, or other properties of the resource. We refer to it as the *Last-By-Domain* approach, or *LBD* for short. We also note that the idea is implemented differently in practice: unless the user navigates to the download directory via the *File Chooser* UI, e.g., if they copy-paste the target path, the browser will not log the directory as the last one for that domain name. Essentially, this means that the actual accuracy may be lower than what is presented in the graphs below. For ease of discussion and comparison, we assume that the user always navigates with the *File Chooser*. As a final note, LBD was never deployed through our plugin; the results are obtained by simulating its behavior on the data provided by the users, knowing how browsers operate. [6]

To measure the performance of *DiDoCtor*, we use a number of metrics reagarding both efficiency and effectiveness. For the efficiency aspect, we report the elapsed time between when the user requested to download a web resource until the classification process was completed, including the time taken to scrape information off the page. Regarding effectiveness, we consider two distinct metrics:

- **Click distance**: Given that the initial problem formulation is based on minimizing `HNC`, counting the number of directories that a user must traverse might be more suitable. We also evaluate the *breadcrumbs* distance, which is a special case of `HNC`, as we followingly discuss.

- **Classification Accuracy**: Since we are using a classifier, measuring its accuracy, i.e., correctly classified instances, seems like a straightforward approach for comparison.

[6]One can validate this by looking through the browser's code.

5.1 Efficiency

We first discuss the efficiency perspective of our approach. Table 2 summarizes the average runtime and standard deviation (per user) required to gather the necessary information, broken down to the time taken by page scraping and classification steps separately.

Table 2: Runtime (in ms) for the download directory suggestion process, implemented as a plugin

User	Scrape		Classify	
	AVG	*SDEV*	*AVG*	*SDEV*
U1	8.54	39.38	13.3	59.45
U2	24.78	1963.06	56.70	5498.13
U3	2.95	7.82	3.07	7.08
U4	3.98	33.49	8.8	70.19
U5	4.31	19.27	11.54	16.52
U6	4.34	17.03	7.78	24.14
Average	8.15	346.67	16.87	945.92

With the exception of user U2, who appears to be an outlier, all values are really small for both page scraping and classification. Even combined, the time taken by both steps to come up with the download directory is well below 100 milliseconds, hardly noticable by the user.

Regarding user U2, who experiences exceedingly high timings, we can identify the following reasons:

- The entire process is executed in JavaScriprt, a dynamic scripting language, executed as a browser plugin, not as a core browser component.

- JavaScript implements associative arrays, i.e. $(key, value)$ pairs, as consecutive memory bytes without any optimization. Therefore, keyword lookup in a dictionary is an $O(m)$ operation, where m is the size of the dictionary.

- In our implementation we did not consider any particular class model for in-memory manipulation. This led to a substantial amount of time being spent on (re-)evaluating regular expressions, which are needed for parsing, so that we could compute the distance between two items. We did not use any memory data structures either, therefore the 1-NN was found by scanning sequentially through the data every time.

- The measurements refer to elapsed wall clock time, meaning that any other process running concurrently directly influences these readings.

All these reasons can easily add up to a considerable overhead, especially when several terms have been extracted as *keywords*, for which the Jaccard distance is used. In respect, we have implemented the same classifier (1-NN) as a standalone application in Java 1.6, executed on a laptop, with Intel Core Duo @1,86GHz and 2GB RAM. This will give us a better view of how long the classification process would have taken, if the classifier was an integral browser component.

Table 3 portrays the runtimes (per user) for the classification process, which is the most time consuming of the two phases, when executed in JavaScript and Java. Clearly, if we had addressed all of the issues we identified earlier, the entire classification process would have gone completely unnoticed in all of the cases.

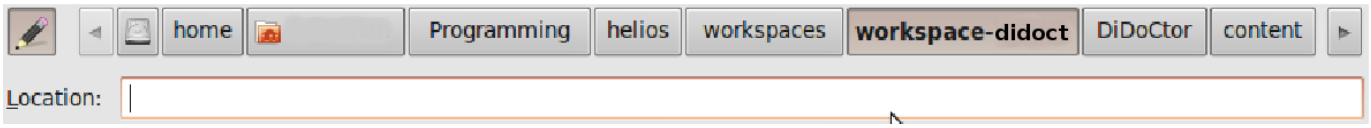


Figure 4: Example of the Breadcrumbs UI

Table 3: Comparative runtimes (in ms) for the download directory suggestion process, implemented as a plugin and as a standalone

User	JavaScript		Java	
	AVG	*SDEV*	*AVG*	*SDEV*
U1	13.3	59.45	9.91	9.3
U2	56.70	5498.13	1.09	1.26
U3	3.07	7.08	0.43	0.50
U4	8.8	70.19	0.74	0.69
U5	11.54	16.52	0.52	0.50
U6	7.78	24.14	0.52	1.06
Average	16.87	945.92	2.2	2.21

For instance, hash-based or binary tree-based [11] implementations of sets would considerably improve lookups for Jaccard computations. In addition, parsing the dataset once at startup, and maintaing it as class objects would ensure that regular expressions are only run once. Finally, such huge time savings would make it possible to gather more information and improve performance or execute more sophisticated algorithms without the user noticing.

5.2 Click Distance

Considering that the initial problem formulation is based on minimizing the navigation cost to the target directory path, we have computed the click distance between the suggested and the target directory, using the HNC measure we discussed in an earlier section. We assume that the cost of going *up* or *down* a directory in the hierarchy is one click. The lower the value of HNC, the better the result, because the user will visit fewer directories until he reaches the desired one.

Table 4: Average click distance (per user) from target directory for *DiDoCtor* and LBD approaches

User	DiDoCtor		LBD		Gain (%)
	AVG	*SDEV*	*AVG*	*SDEV*	
U1	1.97	2.11	2.71	2.25	27.3
U2	0.16	0.72	0.41	1.09	60.9
U3	0.34	0.96	1.58	1.71	78.4
U4	0.69	1.29	1.32	1.81	47.7
U5	0.38	0.78	1.13	1.62	66.3
U6	1.21	1.61	2.06	1.78	41.2
Average	0.79	1.25	1.54	1.71	48.7

Table 4 summarizes the average HNC distance per user, when suggesting a download directory with either approach, and the gain of using *DiDoCtor* over LBD. Clealry, *DiDoCtor* outperforms LBD by far, since in most cases we observe a cutdown by nearly half in the number of clicks performed. The lowest gain is for user U1, who is highly selective in the directories that he uses, and each directory is used a couple of times. Nevertheless, there is still a 27.3% reduction in the number of directories that he will traverse. By contrast, user U3 has an astonishing 78% improvement, bringing the average number of traversed directories close to 0.

More importantly, though, the standard deviation in the number of clicks is the same, if not better (smaller) when using the *DiDoCtor* approach. In other words, our approach consistently requires fewer clicks on average, unlike the LBD approach. Averaging across users, *DiDoCtor* achieves a 48% gain in the directories that users will visit before reaching the right one, which makes our approach more suitable for the problem at hand.

We also computed the click distance in the presence of a *Breadcrumbs* UI. The *Breadcrumbs* UI facilitates navigation between directories, by allowing the user to move to a directory with a single click, provided that:

- The target directory is an ancestor of the one where the user is currently at (i.e., the suggested directory), *or*,
- The target directory is a child directory of the current one, **and** the user has already visited it in the past.

An example of the *Breadcrumbs* UI is shown in Figure 4, with the current directory being: "/home/<user>/Programming/helios/workspaces/workspace-didoct/". The user has already visited the subdirectories "DiDoCtor" and "DiDoCtor/content/". Navigating to a parent directory, e.g., "/home/<user>/ Programming/helios/" is a single click away. However, the user has already paid the cost of navigating to the children directories. In other words, *revisiting* children directories is worth 1 click, but one can not waive the initial cost altogether.

Given this setting, the breadcrumbs click distance is a special case of HNC, where the cost of moving to the common ancestor of the suggested and the target directory is 1, i.e., cost($\mathcal{P}1$, LCA($\mathcal{P}1$, $\mathcal{P}2$))=1. Since the *File Chooser* is always initialized with the suggested directory, none of its children have been visited [7]. Therefore, the user needs to navigate to the correct subdirectory from the common ancestor, and cost(LCA($\mathcal{P}1$, $\mathcal{P}2$), $\mathcal{P}2$) remains unchanged.

Table 5: Average breadcrumbs click distance (per user) from target directory for *DiDoCtor* and LBD approaches

User	DiDoCtor		LBD		Gain (%)
	AVG	*SDEV*	*AVG*	*SDEV*	
U1	1.55	1.55	2.38	2.00	34.9
U2	0.10	0.47	0.38	1.03	73.7
U3	0.26	0.61	1.54	1.67	83.1
U4	0.48	0.90	1.27	1.77	62.2
U5	0.35	0.72	1.10	1.60	68.2
U6	0.86	1.15	1.78	1.60	51.7
Average	0.60	0.90	1.41	1.61	57.4

Table 5 contains the results of the click distance between the two approaches, under the presence of a *Breadcrumbs UI*. Com-

[7] A Breadcrumbs UI that is stateful across downloads may be of interest, but is outside the goals set forth in this paper.

pared to the standard click distance, we observe that all distances are smaller, regardless of technique, which is expected given that the cost of moving to the common ancestor is now always 1. However, the click distance is always more than half of the default HNC distance, which signifies that the higher cost is paid for navigating from the common ancestor to the target directory. *DiDoCtor* still outperforms LBD and the gain is in fact higher, per user and on average. This means that, proportionately, *DiDoCtor* has had a higher improvement in the click distance compared with LBD. Therefore, unlike LBD, the directories suggested by *DiDoCtor* are always in a subtree very close to the target one, demonstrating once more the superiority of our proposed technique.

5.3 Classification Accuracy

We now view the problem from the classification perspective that we cast our initial problem to, reporting on the accuracy of the employed classifier.

Overall accuracy

Figure 5 portrays the average, per-user accuracy, of the classification process, using either technique. The figure basically shows the number of times that the classifier correctly suggested the target directory, where the user saved the resource, during the entire evaluation period. The last column demonstrates the average accuracy across all users, for both techniques.

There are two major conclusions to draw from Figure 5:

1) *DiDoCtor* performs better in all occasions, with substantial improvement in several of them. For example, user U3 experiences a 30% improvement on average, and user U6 approximately 20%. This means that our technique suggests the correct directory right away a lot more frequently than LBD. Combined with the reduced `HNC` cost, *DiDoCtor* emerges as the most suitable approach.

2) Despite the gains that *DiDoCtor* achieves, there are still some cases with objectively low accuracy (e.g., users U1 and U6). This is indicative of the fact that suggesting the right directory where to download web resources is not trivial, and leaves ample room for improvement.

Running average accuracy

To get a more detailed idea of the classification process, we have plotted for each user the classification accuracy as a function of the number of their downloads. For instance, an overall low accuracy could be the result of the user selecting newly created directories, in which case the classifier is destined to fail. Figures 6(a)-(f) demonstrate for each user the classifier's behavior, averaging its accuracy up to the i-th download, for all downloads.

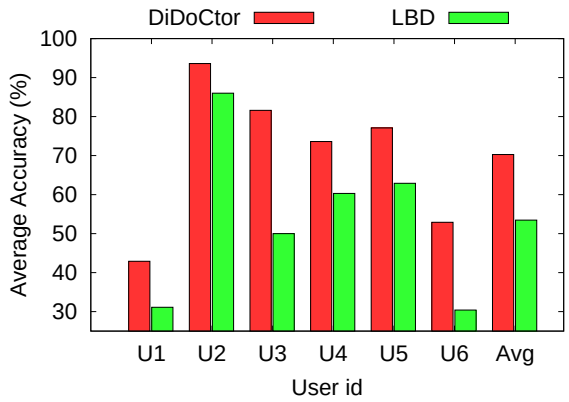


Figure 5: Average per-user accuracy of the classification process

Evidently, the *DiDoCtor* approach is consistently superior to LBD, across all users. The advantages in performance may be obvious (Figure 6(c),(e),(f)) or they may be less prominent (Figure 6(b)). Whichever the case, we observe that unlike LBD, *DiDoCtor* makes good use of the features and is better at suggesting the directory where the resource should be downloaded.

Figure 6(a) shows the running average accuracy of user U1, who is the most selective one, as shown in Figure 2. The two approaches perform almost the same up to the 60-th download, with *DiDoCtor* being better, if only marginally. After that point, LBD continues to deteriorate in performance, whereas *DiDoCtor* is more stable for longer periods of time. Looking through the submitted data, this user downloads resources from sites with varying material, such as online email, and web sites with scientific publications, e.g., **`http://dl.acm.org`**. The resource is then downloaded to different directories, possibly based on publication type, such as *databases*, *user interfaces*, etc., or conference, e.g., *CHI*, *IUI*, *SIGMOD*, etc.

Through manual inspection, we observe that a lot are subdirectories of the same path, e.g., "/0/1/2/3/4/5", "/0/1/2/3/4/6" [8], etc. In several cases, the user selects a sibling node of the one that we suggested, which was not encountered before. Although we do not have actual timestamps for the creation time of these directories, the most probable case is that the user was creating new subdirectories to accommodate downloaded files, which our technique is unable to capture, and also explains the gradual drop in accuracy.

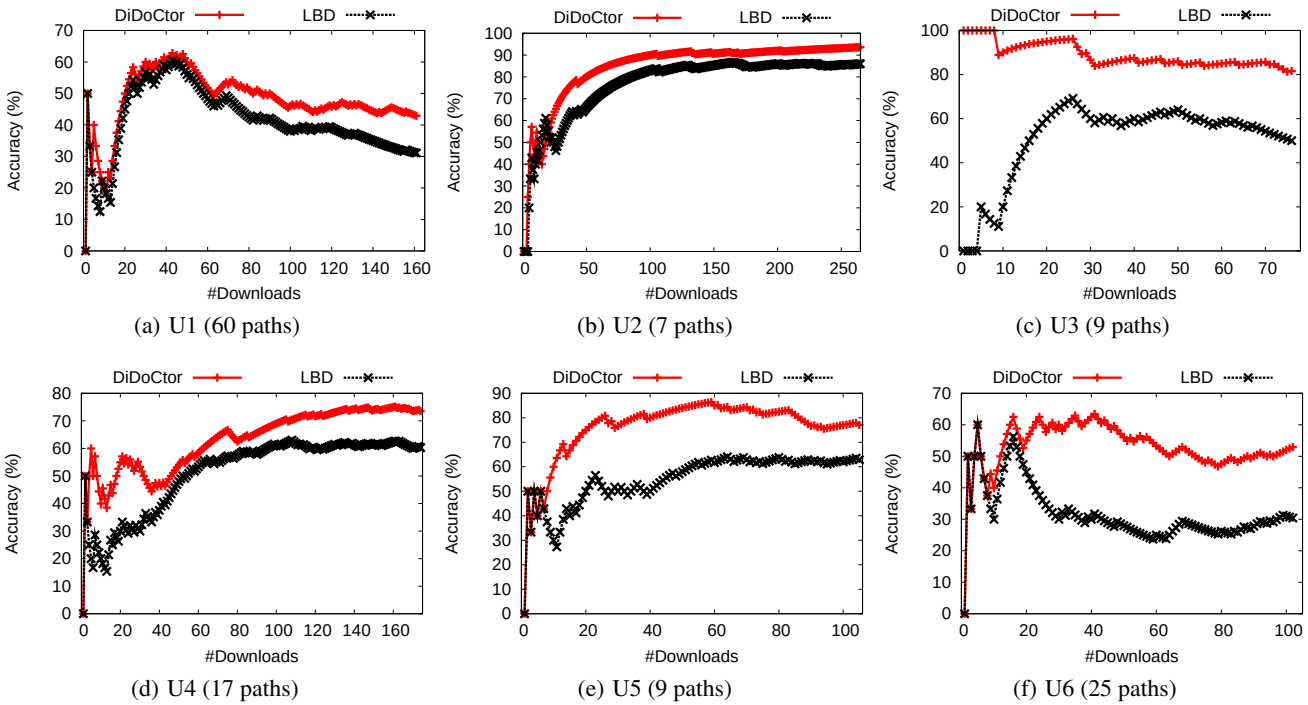
Particularly interesting are users U3 and U5, shown in Figure 6(c) and (e) respectively, who both use 9 directory paths overall. The interesting part is that in both cases, LBD performs poorly with so few paths, achieving (in the long run) an approximate 50-60% accuracy, as depicted by Figure 5 as well. In fact, regarding user U3, it even takes a few downloads before LBD increases in accuracy and levels off at around 60%. On the other hand, *DiDoCtor* achieves at least 80% classification accuracy, meaning that we obtain a minimum 20% improvement. More importantly, high accuracy is obtained early on, with a 70% average for user U5 and as high as 90% average for user U3 from the 20-th download. A user who perceives such good accuracy early on would be more willing to navigate to the target directory than save the files to a temporal location.

The case of user U6 is a clear indication of our claim that *DiDoCtor* makes good use of the features we extracted. The LBD approach has a steep decline in performance after the 20-th download, ending up with 30% accuracy. However, after the 20-th download, *DiDoCtor* deteriorates slightly only temporarily, but then recovers and maintains an average 60% accuracy. Finally, even with user U2, who uses the least number of directories, *DiDoCtor* seems to perform better than LBD. It appears that, around download 25-30, LBD made some erroneous suggestions consecutively, dropping the overall accuracy from 60% down to 45%. On the contrary, *DiDoCtor* managed to predict the right target directory, constantly increasing its performance.

Weight Optimization

Up until now, we have experimented with the custom weights that the plugin was provided with. However, it may well be the case that these weights are sub-optimal, or even bad choices. We want to see how much the effectiveness measures may improve, if we fine-tune

[8]Directory names have been obfuscated before the users submitted their data, to preserve anonymity



(a) U1 (60 paths) (b) U2 (7 paths) (c) U3 (9 paths)

(d) U4 (17 paths) (e) U5 (9 paths) (f) U6 (25 paths)

Figure 6: Running average of classification accuracy

the classifier's weights. We use an equi-weight baseline, where all features contribute equally to the 1-NN classification process. We also employ the `RELIEF_F` algorithm, proposed in [28], which is a weight-optimization technique, based on measuring how similar attributes are with each other among items of the same and different classes. We stress that the idea is not to validate the rationale behind `RELIEF_F`, but rather find out the extent to which we can improve the classification accuracy. Table 6 summarizes the results of this experiment.

Table 6: Average classification accuracy for various 1-NN weights

User	Custom	Equiweight	RELIEF_F
U1	42.9	39.1	44.1
U2	93.6	93.2	94.7
U3	81.6	81.6	81.6
U4	73.6	70.1	76.4
U5	77.1	78.1	83.8
U6	52.9	51.9	53.9
Average	70.28	69.0	72.42

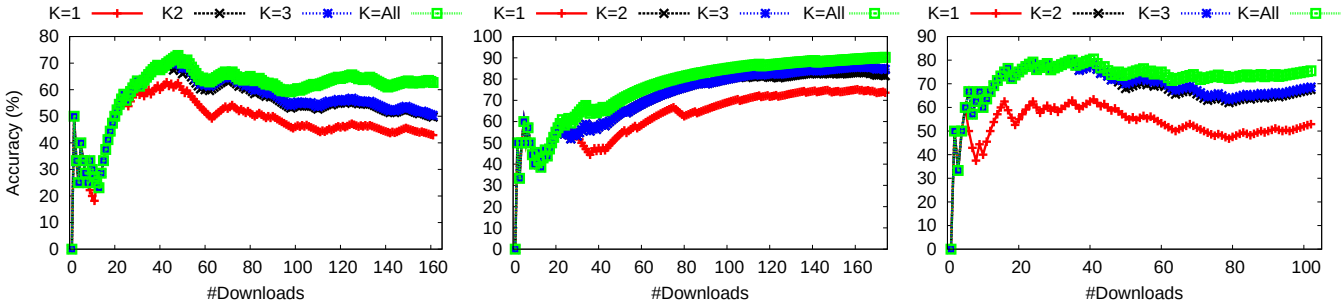
The equi-weight scheme operates worse than our default choice values in all but one occasions. The single exception is user U5, who has a 1% gain with the equi-weight scheme. On the contrary, user U6 would experience a 1% decrease in accuracy. Overall, the equi-weight scheme achieves a lower average by 1.3%. The most important remark, however, is that even with this slight decrease, the equi-weight scheme still behaves better than LBD (per-user and on average), thereby strengthening our claim that a classifier is a much better choice for the download directory suggestion problem.

On the other hand, the `RELIEF_F` approach improves classification accuracy in all occasions but one (user 3), where the accuracy remains unchanged. In particular, the improvement is as low as 1%, e.g., for user U6, but can be as high as 6.7%. Even user U1, who is the most challenging case, experiences a 1.3% improvement overall. All in all, by optimizing the weights, we achieve an average 2% classification accuracy gain across all users. We stress at this point that `RELIEF_F` needs only be run *after* the user selected the download directory. This means that weight optimization can be executed in the background, and will not affect the runtime of the suggestion process.

Table 7: Weights computed by the RELIEF_F algorithm

	U1	U2	U3	U4	U5	U6
time	0.138	0.252	0.242	0.183	0.264	0.212
dname	0.108	0.085	0.089	0.12	0.113	0.129
path	0.107	0.069	0.089	0.062	0.072	0.115
fname	0.069	0.078	0.074	0.043	0.053	0.069
rdname	0.053	0.073	0.09	0.088	0.062	0.053
rpath	0.062	0.06	0.08	0.049	0.018	0.032
rfname	0.047	0.063	0.074	0.042	0.027	0.028
title	0.117	0.08	0.089	0.115	0.08	0.064
filename	0.119	0.086	0.008	0.126	0.168	0.143
filetype	0.094	0.085	0.077	0.074	0.085	0.105
keywords	0.086	0.069	0.088	0.098	0.058	0.05
STDEV	0.029	0.054	0.055	0.043	0.070	0.056

Table 7 shows the weights of the features that we have discussed, as computed by the RELIEF_F algorithm, for each user of our experiment. These weights are the ones obtained after the last download for each user. At the end, we also show the standard deviation of the weights per user, i.e., how much the features differ in their contribution to the user's choices. Time has a high weight in all



(a) U1 (60 paths) (b) U4 (17 paths) (c) U6 (25 paths)

Figure 7: *DiDoCtor* accuracy when suggesting k alternative directories

cases, which means that there is a temporal relation between downloads. The domain name, path, and filename of the current page are also quite important in all cases, and these are the two aspects that LBD tries to combine. However, the name of the file being downloaded contributes quite significantly, except for U3, which is not considered by LBD. Moreover, for user U1 the title is also especially important, whereas the extension contributes greatly for U6. On the other hand, it appears that keywords are not as significant, and the name of the referal contributes even less. Overall, we observe that different users value different groups of features, and to different extents, which validates our claim for a classification-based approach.

Returning k directory paths

An advantage of the MailCat system is that it suggests multiple alternative categories for the user to select while organizing their emails. Similarly, we may suggest more directory paths where the user can download the resource. We can easily modify our 1-NN classifier to return up to k results from different classes, ranked according to their distance from the target item. The user is then shown all k directory paths to select the most suitable. Note that such an approach requires a radical redesign of the *File Chooser* UI, so that more than 1 paths can be displayed simultaneously, in a ranked fashion. However, this issue is outside the scope of this paper, as we are interested in the effectiveness of the technique. A possible solution to this problem can be found in [9].

Due to lack of space, we will focus on the three users with lowest accuracy, users U1, U4 and U6. For each new download, we retrieve its k nearest neighbors, each one from a different directory path. We assume that we successfully suggested the download directory if the user saved the resource in one of the k directories that were displayed. Clearly, suggesting all of the past directories yields the highest possible accuracy, but this is an impractical solution as the UI would be heavily cluttered. Most importantly, though, it is very unlikely that the user will be able to process all that information, considering the "7 ± 2 rule" [30] about the number of items someone is capable of dealing with simultaneously.

Following existing approaches [9, 34], we let $k \leq 3$ and run our 1-NN classifier using the weights obtained from `RELIEF_F`. We also experiment with a k value equal to the total number of directories ever used by the user. Essentially, this serves as an upper bound to the achievable accuracy by any approach that relies on past knowledge. To achieve higher accuracy, the employed approach should be able to suggest *previously unseen* directories, including ones that were just created by the user, which is difficult at best.

Figures 7(a)-(c) portray the results of the above experiment for users U1, U4 and U6, for the different values of k. For $k = 1$, we obtain the same results as in Figures 6(a),(d),(f). Several remarks can be made by looking at these figures. First of all, as expected, accuracy improves in both occasions, when we return more directories. Especially for user U6, there is a 20% improvement compared to suggesting a single directory, raising the accuracy very close to the best attainable (k = ALL). User U4 would also experience higher satisfaction when presented with more directories, with a minimum 10% improvement in the classifier's accuracy. Secondly, the difference in accuracy between $k = 2$ and $k = 3$ is minimal, compared to the difference between $k = 1$ and $k = 2$. This is very important, because it implies that a redesigned *File Chooser* UI would have to make less room, to accommodate 2 instead of 3 directories. We note that both statements also hold for the other 3 users not shown here. Finally, regarding user U1 in particular, it appears that the maximum accuracy we can achieve is around 62%. Taking this into account, *DiDoCtor*'s accuracy of 42% with a single directory is in fact within 67% of the best case scenario.

5.4 Alternative Classifiers

Up to this point, we have only considered variations of the 1-NN classifier that our plugin was distributed with. Therefore, we turn our attention to other classifiers that can be used instead, such as decision trees [33] or SVMs [13]. Note that these techniques are known to be effective for both numerical and categorical data and can be efficiently trained and updated [18, 26]. For this experiment, we considered the implementations of Weka [24]. With the exception of **timestamp**, we converted all features we have already described into boolean vectors through Weka filters. Timestamps were handled as numerical attributes. Although this experiment was performed offline, we simulate the downloading process, had our plugin been distributed with the respective classifier. More precisely, for each download, we perform the classification step to suggest a directory. The actual selected directory is known from the submitted data. The new information of the latest download is used to retrain the classifier. The trained model will be used during the classification process of the next download action.

Figure 8 portrays the average classification accuracy (i.e., at the end of all downloads) of an SVM and a C4.5 classifier, compared with *DiDoCtor* and LBD, to get a sense of their performance. We can make the following remarks:

- Using a classifier always yields better results than LBD, both for users with few (U2) or a lot directories (U6).
- *DiDoCtor* seems to perform the best among all techniques. User U2 is the only exception, where *DiDoCtor* is worse than

SVM and C4.5, but only slightly. On the other hand, *DiDoCtor* outperforms all approaches regarding user U1.

- SVM seems better than C4.5 on some occasions, whereas C4.5 is better on others. The two techniques appear to be somewhat tied on average. Nevertheless, they are both still better than LBD by $\sim$15% on average.

The basic remark from this experiment is that using a classifier is a lot better than the LBD approach, currently employed by most browsers. Browsers already use classifiers for phishing attack [27] countermeasures or web page prefetching [22]. Therefore, integrating a classification-based solution to the directory download problem should be a lot easier to accomplish, and more effective at the same time.

5.5 Additional Remarks

Up till now, our analysis assumed that the user navigates through directories by moving up or down, one directory at a time. However, *File Chooser*s usually have an area with a fixed set of frequently used directories for fast access, such as "Pictures", Music", etc. This means that the actual click cost is in fact lower than the one computed through HNC, even when *Breadcrumbs* are used. However, this does not invalidate our overall analysis:

a) Recall that we addressed the problem as a classification one. Even if the user selected the target directory from the fast access area, the classifier's accuracy would increase if the suggested directory was correct, otherwise it would decrease. As we experimentally demonstrated, *DiDoCtor*'s accuracy was always higher than that of LBD, meaning that we suggested the correct directory a lot more often, regardless of the way that the user reached the target directory.

b) We already showed that *DiDoCtor* can be used to return k directories, achieving even better results. Had we known the contents of the "fast access" area, we could have suggested non-overlapping directories, effectively increasing the value k and, consequently, the classifier's accuracy. This way, we also provide more starting points for the user to navigate and reach the target directory faster.

c) Finally, we acknowledge the fact that a user may search through their directories before selecting the target one, visiting a lot more than what `HNC` mandates. Alternatively, the user may immediately select a directory by copy-pasting its full path. However, whether a user will reach the target directory immediately or after deliberation depends on factors which are unrelated to the resource itself, e.g., the user's current state of mind, their concentration, etc. Such aspects neither approach can tackle, and require extensive experimentation and evaluation, providing ground for further research. We highlight the fact, though, that *DiDoCtor* is able to suggest candidate directories which are really close to the target one, thereby providing a good starting point.

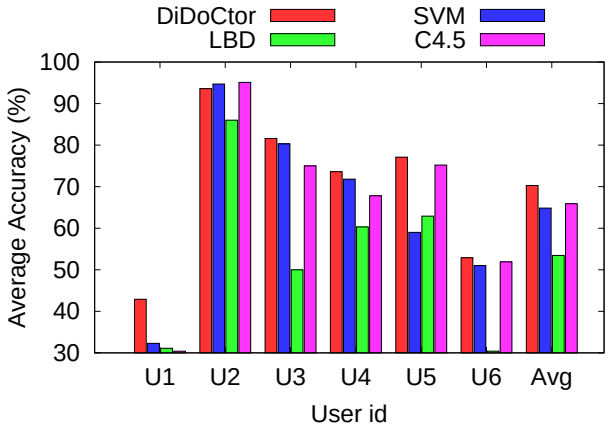


Figure 8: Comparison of classification techniques

6. CONCLUSIONS AND FUTURE WORK

Web browsers are effectively one of the most basic applications in todays' user interaction with computers, and the web in particular. Surveys have shown that downloading web resources such as documents, images, etc., and saving them to a local directory is common among users. However, no particular attention has been paid to this process in helping users automatically organize their downloads contextually. In this paper we tackled this problem, suggesting a good directory where the web resource should be downloaded. We presented a rigid formalism of the problem and then cast it to a classification framework to solve it. Taking into account the efficiency constraints posed by the fact that this is a UI related problem, we presented specific techniques to address it. Our experimental evaluation from real user experience shows a minimum 10% improvement on suggesting the correct directory right away, and a general minimization of the number of clicks needed to navigate to the target directory.

As future work, we plan to investigate further how the temporal dynamics of the downloads can be used to facilitate this process. For instance, statistics regarding the temporal nature of sessions (e.g., average length), could prove particularly useful. Techniques known to boost web search, such as entity extraction and recognition, may also be worth considering. Although our goal is quite apart from web search, the output of these processes could be used as additional features to improve classification accuracy. Extensions also include i) predicting the creation of a directory, ii) suggesting possible names for the web resource, and iii) suggesting the right directory for the reverse process, i.e., uploading an email attachment or submitting a file online.

Acknowledgements

The authors would like to thank the evaluation volunteers for using the plugin and submitting their data. This work has been cofinanced by EU and Greek National funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Programs: Heraclitus II fellowship, THALIS - GeomComp, THALIS - DISFER, ARISTEIA - MMD" and the EU funded project INSIGHT.

7. REFERENCES

[1] Australian Bureau of Statistics. 4. Personal internet use - Table 5, Australia, 2010-11, http://www.abs.gov.au/AUSSTATS/abs@.nsf/DetailsPage/8146.02010-11?OpenDocument, accessed 27 Feb 2014.

[2] Statistics Canada. Internet use by individuals, by type of activity, http://www.statcan.gc.ca/tables-tableaux/sum-som/l01/cst01/comm29a-eng.htm, accessed 29 Sep 2012.

[3] The Radicati Group, Inc., Email Statistics Report 2009-2013, http://www.radicati.com/wp/wp-content/uploads/2009/05/email-stats-report-exec-summary.pdf, accessed 29 Sep 2012.

[4] Save File To. https://addons.mozilla.org/en-us/firefox/addon/save-file-to/, accessed 6 Jan 2013.

[5] Automatic Save Folder. https://addons.mozilla.org/en-US/firefox/addon/automatic-save-folder/, accessed 29 Sep 2012.

[6] Save Link in Folder. https://addons.mozilla.org/en-US/firefox/addon/save-link-in-folder/, accessed 29 Sep 2012.

[7] Previous Folders. https://addons.mozilla.org/en-us/firefox/addon/previous-folders/, accessed 29 Sep 2012.

[8] W3C, HTML 4.01 Specification, http://www.w3.org/TR/html401/, accessed 29 Sep 2012.

[9] X. Bao and T. G. Dietterich. Folderpredictor: Reducing the cost of reaching the right folder. *ACM Trans. Intell. Syst. Technol.*, 2(1):8:1–8:23, Jan. 2011.

[10] Z. Bar-Yossef, I. Keidar, and U. Schonfeld. Do not crawl in the dust: Different urls with similar text. In *WWW*, pages 111–120, 2007.

[11] R. Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf.*, 1:290–306, 1972.

[12] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Comput. Netw. ISDN Syst.*, 29(8-13):1157–1166, Sept. 1997.

[13] C. J. C. Burges. A tutorial on support vector machines for pattern recognition. *Data Min. Knowl. Discov.*, 2(2):121–167, jun 1998.

[14] T. W. Butler. Computer response time and user performance. In *Proc. CHI*, pages 58–62, 1983.

[15] S. Chakrabarti, B. Dom, R. Agrawal, and P. Raghavan. Scalable feature selection, classification and signature generation for organizing large text databases into hierarchical topic taxonomies. *The VLDB Journal*, 7(3):163–178, aug 1998.

[16] G. Dannenbring. System response time and user performance. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-14(3):473–478, may-june 1984.

[17] B. D. Davison. Topical locality in the web. In *Proc. SIGIR*, pages 272–279, 2000.

[18] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. KDD*, pages 71–80, 2000.

[19] M. Dredze, T. A. Lau, and N. Kushmerick. Automatically classifying emails into activities. In *Proc. IUI*, pages 70–77, 2006.

[20] S. Dumais and H. Chen. Hierarchical classification of web content. In *Proc. SIGIR*, pages 256–263, 2000.

[21] N. Eiron and K. S. McCurley. Analysis of anchor text for web search. In *Proc. SIGIR*, pages 459–460, 2003.

[22] D. Fisher and G. Saksena. Link prefetching in mozilla: a server-driven approach. In F. Douglis and B. D. Davison, editors, *Web content caching and distribution*, pages 283–291. 2004.

[23] J. Fürnkranz. Exploiting structural information for text classification on the www. In *Proc. IDA*, pages 487–498, 1999.

[24] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *SIGKDD Explor. Newsl.*, 11(1):10–18, nov 2009.

[25] Infoplease. Most Popular Internet Activities, http://www.infoplease.com/ipa/A0921862.html, accessed 29 Sep 2012.

[26] T. Joachims. Training linear svms in linear time. In *Proc. KDD*, pages 217–226, 2006.

[27] E. Kirda and C. Kruegel. Protecting users against phishing attacks. *Comput. J.*, 49(5):554–561, Sept. 2006.

[28] I. Kononenko. Estimating attributes: analysis and extensions of relief. In *Proc. ECML*, pages 171–182, 1994.

[29] O. A. McBryan. GENVL and WWWW: Tools for taming the web. In *Proc. WWW*, page 15, CERN, Geneva, 1994.

[30] G. A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, (2):81–97, March.

[31] D. Milne and I. H. Witten. An effective, low-cost measure of semantic relatedness obtained from wikipedia links. In *Proc. AAAI*, pages 25–30, 2008.

[32] NielsenWire. What americans do online: Social media and games dominate activity. http://blog.nielsen.com/nielsenwire/online_mobile/what-americans-do-online-social-media-and-games-dominate-activity/, accessed 29 Sep 2012.

[33] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, mar 1986.

[34] R. B. Segal and J. O. Kephart. Mailcat: an intelligent assistant for organizing e-mail. In *Proc. AGENTS*, pages 276–282, 1999.

[35] J. Shen, L. Li, T. G. Dietterich, and J. L. Herlocker. A hybrid learning system for recognizing user tasks from desktop activities and email messages. In *Proc. IUI*, pages 86–92, 2006.

[36] D. Smith. A business case for subsecond response time: Faster is better. In *Computerworld*, pages 1–11, 1983.

[37] S. Stamou, A. Ntoulas, V. Krikos, P. Kokosis, and D. Christodoulakis. Classifying web data in directory structures. In *Proc. APWeb*, pages 238–249, 2006.

[38] D. Sullivan. Top Internet Activities? Search & Email, Once Again, http://searchengineland.com/top-internet-activities-search-email-once-again-88964, accessed 29 Sep 2012.